Query-Based Why-Not Provenance with NedExplain

Nicole Bidoit, Melanie Herschel, Katerina Tzompanaki

Query-Based Why-Not Provenance with NedExplain

Nicole Bidoit
Université Paris Sud / Inria
91405 Orsay Cedex, France
bidoit@lri.fr

Melanie Herschel
Université Paris Sud / Inria
91405 Orsay Cedex, France
melanie.herschel@lri.fr

Katerina Tzompanaki
Université Paris Sud / Inria
91405 Orsay Cedex, France
tzompana@lri.fr

ABSTRACT

With the increasing amount of available data and transformations manipulating the data, it has become essential to analyze and debug data transformations. A sub-problem of data transformation analysis is to understand why some data are not part of the result of a relational query. One possibility to explain the lack of data in a query result is to identify where in the query we lost data pertinent to the expected outcome. A first approach to this so called *why-not provenance* has been recently proposed, but we show that this first approach has some shortcomings.

To overcome these shortcomings, we propose *NedExplain*, an algorithm to explain data missing from a query result. *NedExplain* computes the why-not provenance for monotone relational queries with aggregation. After providing necessary definitions, this paper contributes a detailed description of the algorithm. A comparative evaluation shows that it is both more efficient and effective than the state-of-the-art approach.

Categories and Subject Descriptors

[**Data Curation, Annotation and Provenance**]; [**Data Quality**]

Keywords

data provenance, lineage, query analysis, data quality

1. INTRODUCTION

In designing data transformations, e.g., for data cleaning tasks, developers often face the problem that they cannot properly inspect or debug the individual steps of their transformation, commonly specified declaratively. All they see is the result data and, in case it does not correspond to their intent, developers have no choice but to manually analyze, fix, and test the data transformation again. For instance, a developer may wonder why some products are missing from the result. Possible reasons for such *missing-answers* abound, e.g., were product tuples filtered by a particular selection or are join partners missing? Usually, a developer tests several manually modified versions of the original data transformation that are targeted towards identifying the reason for the missing tuples, for example by removing a selection predicate and observing if the products then appear in the result.

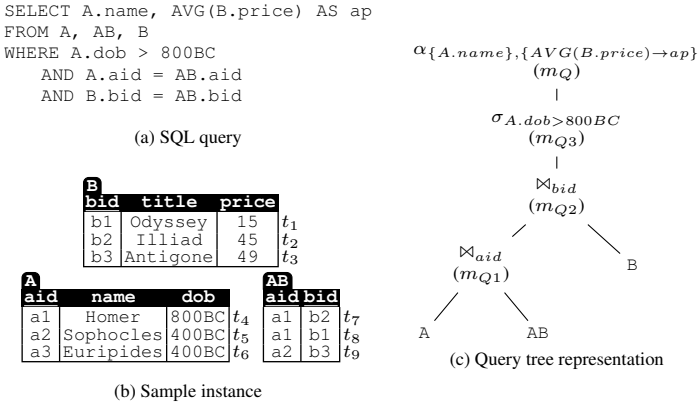


Figure 1: SQL query (a), instance (b), and query tree (c) of running example

To improve on this manual analysis of query behavior and to ultimately help a developer in fixing the transformation, the Nautilus project [13] aims at providing semi-automatic algorithms and tools for query analysis [12], modification, and testing. This paper focuses on the analysis phase, and more specifically, proposes a novel algorithm tackling the sub-problem of explaining missing-answers. Note that explaining missing-answers is not only pertinent for query analysis and debugging, but it also applies to other domains, e.g., to what-if analysis focusing on the behavior of a query.

Explaining missing answers: running example. Very recently, approaches to explain missing-answers of relational and SQL queries have been proposed. This paper focuses on algorithms producing query-based explanations, as illustrated below.

EXAMPLE 1.1. *Consider the SQL query shown in Fig. 1, both in its SQL and query tree form. Ignore the operator labels m_{Q_i} in the query tree for now. Let us further assume the database instance shown in Fig. 1(b). Based on these data and query, the query result includes only one tuple, i.e.,* (*Sophocles, 49*).

Assume that we now wonder why we do not find in the result a tuple with author name Homer and average price greater than 25 (assuming some knowledge on the source data), or more generally, why we do not find any other tuple with a name different from Homer or Sophocles. For this why-not question, two query-based explanations, in the form of picky *subqueries, exist: (1) the selection on attribute* dob *is too strict to let any author named Homer pass (indeed, the* compatible *source tuple* t=(*a1, Homer, 800BC*),

which is a candidate for contributing value `Homer` *to the result, has* `dob = 800BC`*, so the output of the selection contains no* successor *of t) and (2) the join between* `A` *and* `AB` *prunes the only author with name different than Homer or Sophocles.*

Related Work. Our work on query-based why-not provenance falls into the wider research area of *data provenance* and *query debugging*. We briefly review relevant works in this context.

Recently, the problem of relational query and more generally data transformation verification has been addressed by several techniques, including data lineage [5] and data provenance [3], subquery result inspection [9], or visualization [6], or query specification simplification [12, 17, 18]. More generally, methods for debugging declarative programming languages [19] may also apply.

The algorithms computing can be categorized w.r.t. the output they generate. We distinguish between instance-based, query-based, and modification-based why-not provenance.

Instance-based why-not provenance describes a set of source data modifications that lead to the appearance of the missing-answer in the result of a query. In our running example, a possible instance-based result includes the insertion of a tuple (a1, Homer, 801BC) into `A` implying the deletion of (a1, Homer, 800BC) (due to key constraints). Algorithms computing instance-based why-not provenance include Missing-Answers [15] and Artemis [14].

As opposed to that and as illustrated in our running Example 1.1, query-based why-not provenance focuses on finding subqueries responsible for pruning the missing-answer from a query result. The state-of-the-art algorithm computing query-based why-not provenance, called Why-Not algorithm [2], is designed to compute query-based why-not provenance on workflows and also applies to relational queries when considering relational operators as the individual manipulations of the workflow, as presented in [2]. Considering such a workflow and the result it produces w.r.t. a given input, a user may specify a set of tuples missing from the result (so called missing-answers). However, as we will see throughout this paper, the Why-Not algorithm has several shortcomings that may yield incomplete or even incorrect results.

In order to compute modification-based why-not provenance, algorithms [10, 20] rewrite the given SQL query so that the missing-answer appears in the query result of the rewritten query. For instance, in our introductory example, changing the selection condition `A.dob > 800BC` to `A.dob >= 800BC` would result in the inclusion of the answer (Odyssey, 800BC) in the query result, which satisfies the user question.

Very recently, [11] has proposed an algorithm to compute hybrid why-not provenance, a combination of instance-based and query-based why-not provenance. This work is orthogonal to the work presented here, and builds on previous work [14]. The algorithm in [11] may however benefit from any query-based why-not provenance method improving the state of the art and thus from our method *NedExplain*.

Shortcomings of the Why-Not Algorithm. Overall, the shortcomings of [2] are linked to processing queries with self-join, empty intermediate results, the formulation of insufficiently detailed answers, and an inappropriate selection of compatible source data and their successors. We postpone a detailed discussion on these points to Sec.4, as it requires further technical details. However, let us stress two issues based on our running example.

Consider the subquery Q_2 (Fig. 1(c)) of our running example. The output of Q_2 consists of three tuples, e.g., one based on the join between tuples t_4, t_7, and t_2, denoted $t_4t_7t_2$ for short. Hence, the output of Q_2 is $\{t_4t_7t_2, t_4t_8t_1, t_5t_9t_3\}$. Let us consider the question: Why does the output not contain a tuple with the author *Homer* and with a price *49*? Intuitively, we see that the responsible operator is the second join between B and AB, as Homer is not associated to a book with price 49. However, in this case, the Why-Not algorithm [2] does not return any answer. The reason for this is that there are some result tuples with the name Homer ($t_4t_7t_2$, $t_4t_8t_1$), and another one containing a price 49 ($t_5t_9t_3$). So, ignoring that the expected values are not part of the same result tuple, the algorithm comes to the conclusion that the missing result is in fact not missing! From a technical point of view, this is due to the notion of compatible tuples adopted by the Why-Not algorithm (actually called *unpicked data items* in the original publication), which are input tuples that contain pieces of data of the missing answer.

Why-Not [2] may also return inaccurate results, because of the way it traces compatible tuples in the query tree. To illustrate this, let us change the subquery Q_3 of our running example to $\sigma_{A.name=1800}$ and consider the previous Why-Not question on the output of Q_3, which is now empty. To answer the Why-Not question, the Why-Not algorithm [2] identifies as compatible the tuples t_4 from the Authors relation and the tuple t_3 from the Books relation. As we saw before, the output of Q_2 contains the tuples $t_4t_7t_2$, $t_4t_8t_1$, and $t_5t_9t_3$. The two first tuples allow to trace t_4 and the third one to trace t_3. So, the Why-Not algorithm [2] identifies them as $successors$ of the compatible tuples and will continue tracing them until it fails at Q_3 because of the selection. The answer returned by [2] is Q_3. However, as shown before, Homer is not associated to a book with price 49 and so the uppermost join in Q_2 is also responsible for not outputting the desired result although Q_2 is not returned by Why-Not [2]. From a technical point of view, this is due to a too permissive notion of successor tuple.

Contribution. The previous observations w.r.t Why-Not [2] have motivated us to investigate a novel algorithm, named *NedExplain*[1]. Our contribution is:

- *Formalization of query-based why-not provenance.* The current paper provides a formalization of query-based explanations for Why-Not questions that was missing in [2]. It relies on new notions of compatible tuples and of their valid successors. This definition subsumes the concepts informally introduced previously. It covers cases that were not properly captured in [2]. Moreover it takes into account queries involving aggregation (i.e., select-project-join-aggregate queries, or SPJA queries for short) and unions thereof.
- *The* NedExplain *Algorithm.* Based on the previous formalization, the *NedExplain* algorithm is designed to correctly compute query-based explanations given a union of SPJA queries, a source instance, and a specification of a missing-answer within the framework our definitions provide.
- *Comparative evaluation.* The *NedExplain* algorithm has been implemented for experimental validation. Our study shows that *NedExplain* overall outperforms Why-Not, both in terms of efficiency and in terms of explanation quality.
- *Detailed analysis of Why-Not.* We review in detail Why-Not [2] in the context of relational queries and show that it has several shortcomings leading it to return no, partial, or misleading explanations.

Organization. In Sec. 2, we set the theoretical foundation of our algorithm. In Sec. 3, we introduce and discuss *NedExplain* in detail. The experiments and comparative evaluation are presented in Sec. 4. Finally, we conclude in Sec. 5.

[1]The name is inspired by the name of one of the Nautilus' passengers in Jules Verne's novel 20,000 Leagues under the sea, and also stands for non-existing-data-explain.

2. QUERY-BASED EXPLANATION

We assume that the reader is familiar with the relational model [1], and we only briefly revisit relevant notions in our context, in Sec. 2.1. We then formalize the Why-Not question to describe the data missing from a query result, in Sec. 2.2. In Sec. 2.3, we introduce the basic notions necessary to trace data throughout queries, before we more precisely define how we determine the culprit operators. Finally, a formal definition of the why-not answer, i.e., the definition of our query-based why-not provenance, is given in Sec. 2.5.

2.1 Relational Preliminaries

Data model. A *tuple* t is a list of attribute-value pairs of the form $(A_1{:}v_1, \ldots, A_n{:}v_n)$. The *type* of a tuple t, denoted as $type(t)$, is the set of attributes occurring in t. For conciseness, we may omit attribute names when they are clear from the context, i.e., write $(v_1, \ldots, v_n)$.

A *relation schema* of a relation R is specified by $type(R)=\{R.A_1, \ldots, R.A_n\}$. Note that each attribute name A_i in $type(R)$ is qualified by the relation name R.

A *database instance* $\mathcal{I}$ over a database schema $\mathcal{S}=\{R_1, \ldots R_n\}$ is a mapping assigning to each R_i in $\mathcal{S}$, an instance $\mathcal{I}|_{R_i}$ over R_i. For the sake of presentation, we sometimes consider a database instance $\mathcal{I}$ as a set of tuples (of possibly different types).

Queries. As relation schema attributes are qualified, two relation schemas always have disjoint types. To define natural join and union, we thus introduce renaming.

DEFINITION 2.1 (RENAMING ν). *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two disjoint types. A renaming ν w.r.t. $\mathcal{T}_1$ and $\mathcal{T}_2$ is a set of triples (A_1, A_2, A_{new}) where $A_1{\in}\mathcal{T}_1$, $A_2{\in}\mathcal{T}_2$ and $A_{new}{\notin}\mathcal{T}_1 \cup \mathcal{T}_2$ is a new unqualified attribute. The co-domain of a renaming ν, denoted $cod(\nu)$ is the set $\{A_{new} \mid (A_1, A_2, A_{new}){\in}\nu\}$.*

$\mathcal{T}$ being a type, the mapping $\nu(\mathcal{T})$ associates any $A_i \in \mathcal{T}$ to A_{new} if $(A_1, A_2, A_{new}) \in \nu$ and $(A_1{=}A_i) \vee (A_2{=}A_i)$, or otherwise to A_i itself.

With renaming in place, we now define the queries we consider. Essentially, we cover unions of select-project-join (SPJ) or select-project-join-aggregate (SPJA) queries.

DEFINITION 2.2 (QUERY Q). *Let $\mathcal{S}=\{R_1, \ldots, R_n\}$ be a database schema. Then*

1. *$[R_i]$ is a query Q with input schema R_i and target type $type(R_i)$, $i{\in}[1, n]$. $[R_i]$ has no proper subquery.*
2. *Let Q_1, Q_2 be queries with input schemas $\mathcal{S}_1$, $\mathcal{S}_2$, and target types $type(Q_1)$, $type(Q_2)$. Assuming $\mathcal{S}_1{\cap}\mathcal{S}_2{=}\emptyset$:*
   - *$[Q_1] \bowtie_\nu [Q_2]$ is a query Q where ν is a renaming w.r.t. $type(Q_1)$ and $type(Q_2)$. The input schema of Q is $\mathcal{S}_1{\cup}\mathcal{S}_2$. Its target type $\nu(type(Q_1)){\cup}\nu(type(Q_2))$.*
   - *$\pi_W[Q_1]$ where $W \subseteq type(Q_1)$, is a query Q with input schema $\mathcal{S}_1$ and target type W.*
   - *$\sigma_C[Q_1]$ where C is a condition over $type(Q_1)$, is a query Q with input schema $\mathcal{S}_1$ and target type $type(Q_1)$.*
3. *Let Q_1 be a query according to (1) and (2) and $G \subseteq type(Q_1)$. Let also $F{=}\{f_i(A_1) \to A'_1, \ldots, f_n(A_n) \to A'_n\}$ be a list of aggregation function calls with $f_i \in \{sum, count, avg, min, max\}$ and $A_i \in type(Q_1)$ that are associated with the new attribute names A'_i, and let $Agg{=}\{A'_1, \ldots, A'_n\}$. Then, $\alpha_{G,F}[Q_1]$ is a query Q with input schema $\mathcal{S}_1$ and target type $G \cup Agg$.*
4. *$[Q_1] \cup_\nu [Q_2]$ is a query Q where ν is a renaming w.r.t. $type(Q_1)$ and $type(Q_2)$ if $\nu(type(Q_1)){=}\nu(type(Q_2))$, and Q_1 and Q_2 are queries according to (1), (2), and (3). The input schema of Q is $\mathcal{S}_1{\cup}\ \mathcal{S}_2$ and its target type is $\nu(type(Q_1))$.*

For instance, in our example the join $\bowtie_{aid}$ (see m_{Q1} in Fig. 1(c)) stands for $\bowtie_\nu$. The renaming ν in this case is the triple $(A.aid, BA.aid, aid)$ which maps the qualified attributes $A.aid$ and $BA.aid$ to the new attribute aid.

To simplify our discussion, we assume that subqueries are named and that two subqueries Q_1 and Q_2 have distinct target attributes. Of course, users can write their queries in a less restrictive way using traditional SQL syntax, which are then automatically translated to our query form.

Given a unary (binary) query Q built from queries Q_1 (and Q_2), we consider as Q's subqueries Q_1 (and Q_2) as well as their respective subqueries. That is, using a standard tree representation of queries, one node corresponds to a subquery.

We further define an input instance for a query Q whose input schema is $\mathcal{S}_Q$, as an instance over $\mathcal{S}_Q$. This definition, given below, introduces η_Q to correctly deal with self-joins.

DEFINITION 2.3 (QUERY OVER A DATABASE). *A query over a database schema $\mathcal{S}$ is a pair (Q, η_Q) where Q is a query with input schema $\mathcal{S}_Q$ and η_Q is a mapping from $\mathcal{S}_Q$ to $\mathcal{S}$ such that for any $R{\in}\mathcal{S}$, $R.A \in type(R)$ iff $\eta_Q(R).A \in type(\eta_Q(R))$.*

Given an instance $\mathcal{I}$ over $\mathcal{S}$, the evaluation of (Q, η_Q) over $\mathcal{I}$ is defined as the evaluation of Q over the input instance $\mathcal{I}_Q$ over $\mathcal{S}_Q$ defined by: for any $S{\in}\mathcal{S}_Q$, $\mathcal{I}_Q|_S{=}\mathcal{I}|_R$ if $\eta_Q(S){=}R$.

Given the data and query as formalized above, we now formalize how Why-Not questions are expressed.

2.2 The Why-Not Question

Intuitively, we specify the why-not question by means of a predicate characterizing the data which is missing from a query result. Such a predicate is a disjunction of *conditional tuples*, which are essentially attribute-value/variable pairs possibly constrained by conjunctive predicates. We start by introducing tuples with variables and then conditional tuples.

DEFINITION 2.4 (v-TUPLE). *Let V be an enumerable set of variables. A v-tuple t_v of type $\{A_1, \ldots, A_n\}$ is of the form $(A_1{:}e_1, \ldots, A_n{:}e_n)$ where $e_i \in V \cup dom(A_i)$ for $i \in [1, n]$ and $dom(A_i)$ denoting the active domain of A_i.*

The variables of a v-tuple are similar in spirit to labeled nulls, used for instance in the context of data exchange [7]. Intuitively, the semantics associated to such variables is that we do not care about the value of the corresponding attribute.

In general, we want to be able to express that, although the actual value is unknown, it yet should satisfy some constraints. For this reason, we resort to conditional tuples (or c-tuples for short), previously introduced for incomplete databases [16].

DEFINITION 2.5 (CONDITIONAL TUPLE (c-TUPLE)). *Let t_v be a v-tuple and let X be the set of variables in t_v. A* c-tuple *t_c is a pair $(t_v, cond)$ where $cond{=}\bigwedge_{i=1}^{n} pred_i$ and for $1 \leq i \leq n$*

$$pred_i :: true \mid x_1\ cop\ x_2 \mid x_1\ cop\ a$$

where x_i is a variable in X, $a{\in}dom(type(x_1))$, and cop is a comparison operator ($\neq, =, <, >, \geq, \leq$).

The type of a c-tuple $(t_v, cond)$ is the type of t_v. We now are ready to define what is a Why-Not question.

DEFINITION 2.6 (WHY-NOT QUESTION). *A Why-Not question w.r.t. a query Q is a predicate $\mathcal{P}$ over Q's target type $\mathcal{T}_Q$, where $\mathcal{P} = \bigvee_{i=1}^{n} t_c^i$, with t_c^i being a c-tuple s.t. type$(t_c^i) \subseteq \mathcal{T}_Q$.*

EXAMPLE 2.1. *The Why-Not question expressed in Ex. 1.1 corresponds to the predicate $\mathcal{P}$=((A.name:Homer, ap:x_1), $x_1 > 25$) $\vee$ ((A.name:x_2), $x_2 \neq$ Homer$\wedge$ $x_2 \neq$ Sophocles) .*

In the sequel, we will omit the condition when it is $true$, i.e., we may rewrite the c-tuple $(t, true)$ as t. Also, we consider only conditions that compare variables with constants or with variables that are local to the same relation.

As a reminder, given a query Q whose input schema is $\mathcal{S}_Q$, new attributes may have been introduced through join or union specifications. These new attributes are well identified and linked to the input attributes through the renamings used by joins and unions in Q. Answering a Why-Not question requires to trace back tuples belonging to the query input instance, which is an instance over $\mathcal{S}_Q$. This further entails that the c-tuples of the (predicate specifying the) Why-Not question need to be rewritten using attributes in $\mathcal{S}_Q$ only. This translation is done by inversing the query renamings as follows.

DEFINITION 2.7 (UNRENAMED PREDICATE W.R.T. A QUERY Q). *Let t_c be a c-tuple and a ν be a renaming. Given any $(A_1, A_2, A_{new}) \in \nu$, if $A_{new} \in$ type(t_c), we replace each A_{new} in t_c by A_1, denoted as $\nu_{|1}^{-1}(t_c)$. We proceed analogously for A_2, yielding $\nu_{|2}^{-1}(t_c)$.*
Now, let Q be a query. The mapping UnR_Q associates to t_c a predicate defined by:

1. *if $Q = [R_i]$ then $UnR_Q(t_c) = t_c$,*
2. *Let Q_1, Q_2 be queries*
   - *if $Q = [Q_1] \bowtie_\nu [Q_2]$, then $UnR_Q(t_c) = UnR_{Q_1}(\nu_{|1}^{-1}(t_c)) \bowtie UnR_{Q_2}(\nu_{|2}^{-1}(t_c))$*
   - *if $Q = [Q_1] \cup_\nu [Q_2]$, then $UnR_Q(t_c) = UnR_{Q_1}(\nu_{|1}^{-1}(t_c)) \vee UnR_{Q_2}(\nu_{|2}^{-1}(t_c))$*
   - *if $Q = \pi_W[Q_1]$, $Q = \alpha_{G,F}(Q_1)$, or $Q = \sigma_C[Q_1]$ then $UnR_Q(t_c) = UnR_{Q_1}(t_c)$.*

If $\mathcal{P}$ is the predicate $\bigvee_{i=1}^{n} t_c^i$, then the unrenamed predicate associated with $\mathcal{P}$ given the query Q is $\bigvee_{i=1}^{n} UnR_Q(t_c^i)$.

EXAMPLE 2.2. *Assume that our sample query Q includes one more output attribute, i.e., $\mathcal{T}_Q$=$\{A.name, aid, ap\}$, and assume the renaming ν=$\{(AB.aid, A.aid, aid)\}$. For the predicate $\mathcal{P}$=(A.name:Homer, aid:a1, ap:x_1), the attribute* aid *can be unrenamed to $A.aid$ and to $AB.aid$, two qualified attributes that cannot be further unrenamed. So, the unrenamed predicate $\mathcal{P}$ is* (A.name:Homer, A.aid:a1, ap: x_1) $\bowtie$ (A.name:Homer, AB.aid:a1, ap:x_1)=(A.name:Homer, A.aid:a1, AB.aid:a1, ap:x_1).

2.3 Compatibility

Given a Why-Not question about the query Q in the form of a predicate $\mathcal{P}$, we compute the Why-Not answer by tracing source data relevant to the satisfaction of $\mathcal{P}$ through all subqueries of the query Q. We identify such relevant data based on their *compatibility* with $\mathcal{P}$.

DEFINITION 2.8 (c-TUPLE COMPATIBILITY). *Let $\mathcal{I}$ be an instance over a schema $\mathcal{S}$. Let also t_c be a c-tuple with type$(t_c) \subseteq \bigcup_{R \in S}$ type$(R) \cup Agg$, where Agg is defined as in Def. 2.2-3.*
The tuple t=$(R.A_1$:$v_1, \ldots, R.A_n$:$v_n) \in \mathcal{I}|_R$, where $R \in \mathcal{S}$, is compatible *with t_c if, for the unrenamed form of t_c, (1) type$(t) \cap$ type$(t_c) \neq \emptyset$ and (2) there exists a valuation ν for t_c s.t. (a) $\forall A \in$type$(t_c) \cap$ type(t):$\nu(t_c.A)$=$t.A$, and (b) $\nu(t_c) \models t_c.cond$.*
The tuple t is compatible with a predicate $\mathcal{P}$ if it is compatible with at least one c-tuple t_c of $\mathcal{P}$.

EXAMPLE 2.3. *The compatible tuple w.r.t. the c-tuple $t_{c1} =$ ((Homer,x_1), $x_1 > 25$) of our Why-Not question of Ex. 2.1 is $t_4 \in \mathcal{I}_{|A}$ (see Fig. 1(b)). Indeed, both t_{c1} and t_4 have equal values for their shared attribute A.name, and there exists a value for x_1 satisfying $x_1 > 25$.*

The set of tuples compatible with t_c, called *direct compatible set* w.r.t. t_c is denoted by Dir^{t_c}. Let $\mathcal{S}^{t_c}$ be the set of relation schemas typing the tuples of Dir^{t_c}. The *indirect compatible set* w.r.t. t_c, denoted $InDir^{t_c}$, is the restriction of $\mathcal{I}$ over the schema $\mathcal{S}_Q - \mathcal{S}^{t_c}$, thus $Dir^{t_c} \cap InDir^{t_c}$=$\emptyset$.

EXAMPLE 2.4. *Pursuing Ex. 2.3, $Dir^{t_{c1}}$=$\{t_4\}$ whereas $InDir^{t_{c1}}$=$\mathcal{I}_{|AB} \cup \mathcal{I}_{|B}$.*

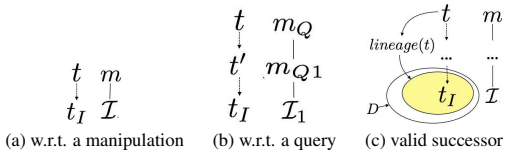
2.4 Pickyness

Intuitively, given a query Q and the set of compatible tuples (both direct and indirect) in $\mathcal{I}_Q$, our goal is to trace compatible tuples in the data flow of the query tree; that is, identify subqueries of Q that destroy successors (formally defined below) of these tuples.

To trace compatible tuples through subqueries, we potentially need to process each subquery in Q one after the other. To formalize this procedure, we associate to each subquery Q_i a manipulation m_{Q_i} that serves as a type signature of Q_i. For instance in Fig. 1, the subquery Q_1 is associated to m_{Q1}, a manipulation of the form $A \bowtie AB$. The input instance $\mathcal{I}_i$ of a manipulation m_{Q_i} includes solely the output of its direct children in the tree (or, in case of leaf nodes, the instance of the corresponding table), e.g. m_{Q_1} and B in Fig. 1 for m_{Q_2}. The output of a manipulation m over its input instance $\mathcal{I}$ is denoted by $m(\mathcal{I})$.

Data lineage, or lineage for short as defined in [4], is at the basis of tuples tracing. Because of space limitation, we cannot reproduce the formal definition of lineage of [4]. The purpose of the next example is to give the intuition of how lineage is defined for operators and also to explain our notation. Next we consider two relation schemas $R(A, B)$ and $S(A, B)$ and the database instance $\mathcal{I}$=$\mathcal{I}_R \cup \mathcal{I}_S$ where $\mathcal{I}_R$=$\{(a_1, b_1), (a_1, b_2), (a_2, b_1)\}$ and $\mathcal{I}_S$=$\{(a_1, b_1), (a_2, b_2)\}$. Let us consider the union operator within the manipulation m=$[R] \cup [S]$ whose evaluation on $\mathcal{I}$ produces $m(\mathcal{I})$=$\{(a_1, b_1), (a_1, b_2), (a_2, b_1), (a_2, b_2)\}$. First note that the lineage of t is only defined when $t \in m(\mathcal{I})$. So, the lineage of the tuple t=(a_1, b_1) w.r.t. m and $\mathcal{I}$, is defined in [4] as a tuple of instances $lineage(t)$=$<\mathcal{J}_R, \mathcal{J}_S>$, where $\mathcal{J}_R$=$\{(a_1, b_1)\}$ is an instance over R and $\mathcal{J}_S$=$\{(a_1, b_1)\}$ an instance over S. In our setting, the lineage of the tuple t is exactly the same although $lineage(t)$ is presented as a set of tuples (typed tuples) i.e. $lineage(t)$=$\{(R.A : a_1, R.B : b_1), (S.A : a_1, S.B : b_1)\}$.

Given a manipulation m and an input instance $\mathcal{I}$, we define that $t \in m(\mathcal{I})$ *is a successor of* some $t_\mathcal{I} \in \mathcal{I}$ by $t_\mathcal{I}$ is in the lineage of t w.r.t. m. Conversely, we say that $t_\mathcal{I}$ is a *predecessor* of t. Fig. 2(a) illustrates the successor relationship between t and $t_\mathcal{I}$ belonging to $m(\mathcal{I})$ and $\mathcal{I}$, respectively.



(a) w.r.t. a manipulation (b) w.r.t. a query (c) valid successor

Figure 2: Successor t of a tuple $t_\mathcal{I}$

We now define the notion of tuple successor w.r.t. to a composed query. Note that in the following definition, UOp is a unary operator among σ, π, α, and BOp is a binary operator among $\cup$, $\bowtie$. The definition is illustrated in Fig. 2(b) for the case of unary operators.

DEFINITION 2.9 (TUPLE SUCCESSOR W.R.T. A QUERY). *Let Q be a query over $\mathcal{S}_Q$ and $\mathcal{I}$ be an instance over $\mathcal{S}_Q$. A tuple $t \in Q(\mathcal{I})$ is a successor of some $t_\mathcal{I} \in \mathcal{I}$ w.r.t. Q if, for $Q = UOp[Q_1]$ (resp. $Q=[Q_1]BOp[Q_2]$), there exists some $t' \in Q_1(\mathcal{I}_1)$ (resp. $t' \in Q_1(\mathcal{I}_1) \cup Q_2(\mathcal{I}_2)$) such that t is a successor of t' w.r.t. m_Q and $Q_1(\mathcal{I}_1)$ (resp. $Q_1(\mathcal{I}_1) \cup Q_2(\mathcal{I}_2)$) and either $t'=t_\mathcal{I}$ or t' is a successor of $t_\mathcal{I}$ w.r.t. Q_1 (resp. Q_1 or Q_2). Here, $\mathcal{I}_i$ is the instance over $\mathcal{S}_{Q_i}$ defined by $\mathcal{I}_i = \mathcal{I}\,|_{\mathcal{S}_i}$ for $i=1,2$.*

We now restrict the notion of successors to *valid* successors w.r.t. some tuple set D. This restriction demands that the lineage of a tuple successor is fully contained in D. In practice, D corresponds to all compatible tuples (direct and indirect) and is used to ensure the correctness of our Why-Not answers.

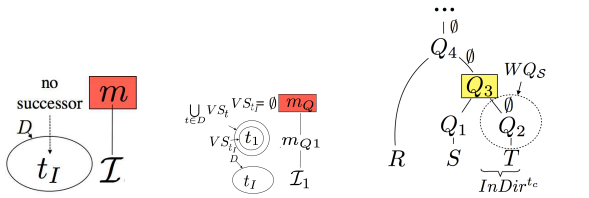
NOTATION 2.1 (VALID SUCCESSOR). *Let Q be a query, $\mathcal{I}$ be a well typed input instance for Q and $D \subseteq \mathcal{I}$. A tuple $t \in Q(\mathcal{I})$ is a valid successor of some $t_\mathcal{I} \in D \subseteq \mathcal{I}$ w.r.t. Q if t is a successor of $t_\mathcal{I}$ w.r.t. Q and $lineage(t) \subseteq D$.*
Next, $VS(Q, \mathcal{I}, D, t)$ denotes, for a given instance $\mathcal{I}$, the set of valid successors of $t \in D \subseteq \mathcal{I}$ w.r.t. Q.

In our running example, consider the subquery Q_2 on the input instance $\mathcal{I}$ shown in Fig.1(b). Then let $D=\{t_4, t_2\} \cup \mathcal{I}_{|AB}$ and consider the tuple $t_4 \in D$. The output of Q_2 is $Q_2(\mathcal{I})=\{t_4t_7t_2, t_4t_8t_1, t_5t_9t_3\}$ (each output tuple is represented by the identifiers of the tuples in its lineage). We say that the output tuple $t_4t_7t_2$ is a *valid* successor of t_4 because it is a successor of t_4 w.r.t. Q_2 and $\mathcal{I}$ and its lineage is included in D (i.e., t_4, t_7, t_2 are all in D). On the contrary, the output tuple $t_4t_8t_1$ is not a *valid* successor of t_4 even though it is a successor of t_4, because the tuple t_1 which is in the lineage of $t_4t_8t_1$, is not in D.
Fig. 2 illustrates the notion of valid successor. From now on, we will generally refer to valid successors when writing successor, unless mentioned otherwise.

When tracing tuples - more specifically, compatible tuples - throughout the query, our goal is to identify which subqueries are responsible for "losing" compatible tuples. These are declared as *picky*, a property at the heart of our definition of Why-Not answers. More specifically, we define picky manipulations and subqueries w.r.t. a tuple set D and a tuple $t_\mathcal{I} \in D$. The definitions, given below, are illustrated in Fig. 3.

DEFINITION 2.10 (PICKY MANIPULATION). *Let m be a manipulation, $\mathcal{I}$ be a well typed input instance for m and $D \subseteq \mathcal{I}$. Then m is a picky manipulation w.r.t. D and $t_\mathcal{I} \in D$, if there is no valid successor t of $t_\mathcal{I}$ in $m(\mathcal{I})$.*

DEFINITION 2.11 (PICKY QUERY). *Let Q be a query over $\mathcal{S}_Q$, $\mathcal{I}$ an input instance for Q and $D \subseteq \mathcal{I}$ a set of tuples. Let $t_\mathcal{I}$ be a tuple in D.*
Assuming that $Q=[Q_1]BOp[Q_2]$ and that $t_\mathcal{I} \in \mathcal{I}_1$ (the case of $t_\mathcal{I} \in \mathcal{I}_2$ is dual), Q is picky w.r.t. D and $t_\mathcal{I}$ if



(a) Picky manipulation (b) Picky query (c) Secondary Why-Not answer

Figure 3: Pickyness ((a)&(b)) and secondary Why-Not answer (c)

1. $VS(Q_1, \mathcal{I}, D, t_\mathcal{I}) \neq \emptyset$
2. *for each $t_1 \in VS(Q_1, \mathcal{I}, D, t_\mathcal{I})$, m_Q is picky w.r.t. the tuple t_1 and the set $\bigcup_{i=1,2} \bigcup_{t \in D} VS(Q_i, \mathcal{I}, D, t)$ considering the input instance $\bigcup_{i=1,2} Q_i(\mathcal{I}_i)$.*

Now, assuming that $Q=UOp[Q_1]$ and that $t_\mathcal{I} \in \mathcal{I}_1$, Q is picky w.r.t. D and $t_\mathcal{I}$ if

1. $VS(Q_1, \mathcal{I}, D, t_\mathcal{I}) \neq \emptyset$
2. *for each $t_1 \in VS(Q_1, \mathcal{I}, D, t_\mathcal{I})$, m_Q is picky w.r.t. the tuple t_1 and the set $\bigcup_{t \in D} VS(Q_1, \mathcal{I}, D, t)$ considering the input instance $Q_1(\mathcal{I}_1)$.*

Note that in the definition of a picky query, item 1 enforces that, just before the top level operator of Q, the tuple $t_\mathcal{I}$ could still be traced and item 2 determines that it is no more the case for the top level operator of Q.
It is easy to prove that the following property holds.

PROPERTY 2.1. *Let Q be a query over $\mathcal{S}_Q$ and let $\mathcal{I}$ be an instance over $\mathcal{S}_Q$. Let also $D \subseteq \mathcal{I}$ be a set of tuples and $t_\mathcal{I} \in D$. Then, there exists at most one subquery Q' of Q, s.t. Q' is picky w.r.t. D and $t_\mathcal{I}$.*

EXAMPLE 2.5. *For $t_{c1}=((Homer, x_1), x_1>25)$, assume $D=\{t_4\} \cup \mathcal{I}_{AB} \cup \mathcal{I}_B$. Q_1 has two valid successors of t_4, i.e., those joining t_4 with $t_7 \in \mathcal{I}_{AB} \subseteq D$ and $t_8 \in \mathcal{I}_{AB} \subseteq D$, respectively. Similarly, Q_2 has two valid successors of t_4, their respective lineage $\{t_4, t_7, t_2\}$ and $\{t_4, t_8, t_1\}$ being in D. Finally, we observe that t_4 has no (valid) successor w.r.t. Q_3 because t_4 does not satisfy the selection condition $A.dob > 800BC$. Therefore, Q_3 is picky w.r.t. t_4 and D.*

2.5 Why-Not Answers

In this section, we provide three kinds of answers for a Why-Not question specified wrt a query Q by a single unrenamed c-tuple t_c. These answers differ in terms of their level of detail or of their point of view. They are based on the notion of picky subqueries and consider the direct compatible set Dir^{t_c} and the indirect compatible set $InDir^{t_c}$. When a Why-Not question is expressed in the form of a Predicate $\mathcal{P}$, i.e., a disjunction of compatible tuples, the Why-Not answer of $\mathcal{P}$ is the union of the answers of each t_c in $\mathcal{P}$.

For the purpose of the following definitions and of covering aggregation, we assume a subquery (a view) V of Q such that $type(V) \supseteq \{G\} \cup \{A_1, \ldots, A_n\}$ (see Def. 2.2). We defer the discussion of determining V to Sec. 3.1. Intuitively, the output schema of V is such that we can apply the aggregation operator (if present in Q) directly on the view V (as well as on all its ancestors in the query tree), enabling us to verify if the conditions defined by t_c on aggregated values, denoted $t_c.cond_\alpha$, are satisfied.

In the next definitions, we assume that Q is a query over $\mathcal{S}_Q$ and $\mathcal{I}$ is an input instance for Q. We also assume that t_c is an unrenamed conditional tuple, as stated before.

Let us start by defining the detailed answer of a Why-Not question, which records: (1) the picky query per compatible tuple (if any) and (2) in the case of aggregation, the subquery violating the conditions on the aggregated values.

DEFINITION 2.12 (DETAILED WHY-NOT ANSWER). *The detailed Why-Not answer of t_c w.r.t. Q and $\mathcal{I}$, denoted $dW_Q^{\mathcal{I}}(t_c)$, is defined as follows given that Q' below is a unary query of the form $UOp[Q_1]$ (resp. a binary query of the form $[Q_1]BOp[Q_2]$):*

$$\bigcup_{t_{\mathcal{I}} \in Dir^{t_c}} \{(t_{\mathcal{I}}, Q') \mid \begin{array}{l} Q' \text{ subquery of } Q \text{ and} \\ Q' \text{ picky w.r.t. } Dir^{t_c} \cup InDir^{t_c} \text{ and } t_{\mathcal{I}}\end{array}\}$$

$$\cup \quad \{(\bot, Q') \mid \begin{array}{l} V \text{ proper subquery of } Q' \text{ and} \\ Q'_1(\mathcal{I}) (\text{resp. } Q'_1(\mathcal{I}) \cup Q'_2(\mathcal{I})) \models t_c.cond_\alpha \text{ and} \\ Q'(\mathcal{I}) \not\models t_c.cond_\alpha\}\end{array}$$

The second part of this definition ensures that the conditions on aggregated values are verified on the input of Q', but not on its output.

EXAMPLE 2.6. *In our running example $V{=}Q_2$. The detailed Why-Not answer for the first part of our Why-Not question (i.e., $t_{c1} = ((\textsf{Homer}, x_1), x_1 > 25)$) is $\{(t_4, Q_3)\}$ as Q_3 is picky w.r.t. t_4 and $\{t_4\} \cup \mathcal{I}_{|AB} \cup \mathcal{I}_{|B}$ and the data provided by V may satisfy the aggregation condition (e.g., applying the aggregation on the tuples present in V yields an average price of 30, which is above 25), whereas the empty output of Q_3 does not satisfy this condition.*

In general, this detailed answer may be too overwhelming for a user (due to the potentially large number of picked compatible tuples). Thus, we also define a condensed Why-Not answer that only provides the set of picky subqueries to the user, e.g., $\{Q_3\}$ in the previous example.

DEFINITION 2.13 (CONDENSED WHY-NOT ANSWER). *The condensed Why-Not answer for t_c w.r.t. Q and $\mathcal{I}$ is defined as $dcW_Q^{\mathcal{I}}(t_c) = \{Q' | (d_{\mathcal{I}}, Q') \in dW_Q(t_c)\}$.*

Finally, we also define a secondary Why-Not answer that considers the indirect compatible set $InDir^{t_c}$. Recall that $InDir^{t_c}$ includes data necessary to produce t_c, but that is not constrained by t_c (i.e., its necessity is only imposed by Q). Consequently, missing t_c may also be caused by the "disappearance" of $InDir^{t_c}$, a case we capture with the secondary Why-Not answer.

EXAMPLE 2.7. *Assume we replace the right child of Q_2, i.e., B, with the subquery $Q'_1{=}B \bowtie_{bid} TOC$ and $\mathcal{I}_{TOC}{=}\emptyset$. Clearly, $Q'_1(\mathcal{I}){=}\emptyset$. Now, we find (t_4, Q_2) as detailed Why-Not answer w.r.t. t_c and $D{=}\{t_4\} \cup \mathcal{I}_{AB} \cup \mathcal{I}_B \cup \mathcal{I}_{TOC}$. However, the fact that Q_2 picks t_4 may result from the empty result of Q'_1, so we return $\{Q'_1\}$ as secondary Why-Not answer.*

As a reminder, S^{t_c} is the set of relation schemas typing the tuples in Dir^{t_c} and thus $\mathcal{S}_Q - \mathcal{S}^{t_c}$ is the set of relation schemas typing the tuples in $InDir^{t_c}$.

DEFINITION 2.14 (SECONDARY WHY-NOT ANSWER). *Let $\mathcal{S} \in \mathcal{S}_Q - \mathcal{S}^{t_c}$. We denote by $Q_{\mathcal{S}}$ the subquery of Q s.t. $Q_{\mathcal{S}}$ is picky w.r.t. $\mathcal{I}$ and some $d \in \mathcal{I}_{|\mathcal{S}}$, and for any $d' \in \mathcal{I}_{|\mathcal{S}}$, there is no successor of d' w.r.t. $Q_{\mathcal{S}}$. Then, the secondary Why-Not answer of t_c w.r.t. Q and $\mathcal{I}$ is $sW_Q^{\mathcal{I}}(t_c) = \{Q_{\mathcal{S}} \mid \mathcal{S} \in \mathcal{S}_Q - \mathcal{S}^{t_c}\}$.*

Fig. 3(c) illustrates the secondary Why-Not answer.

3. THE NEDEXPLAIN ALGORITHM

Based on the framework our definitions provide, we now present *NedExplain*, an algorithm that takes as input a predicate $\mathcal{P}$ over the output type $type(Q)$ of a query Q over a database schema $\mathcal{S}_Q$ and the query input database instance $\mathcal{I}_Q$. We limit Q to a union of SPJA queries, deferring the addition of further operators to future work. *NedExplain* supports the computation of any type of Why-Not answer (see Sec. 2.5). However, due to the limited space, we focus our discussion on outputting detailed Why-Not answers.

3.1 Preprocessing

NedExplain starts with a preprocessing phase, consisting of the steps described below:

1) Unrenaming. First, we unrename $\mathcal{P}$ as defined in Def. 2.7. Thereby, we obtain $unR(\mathcal{P}){=}\bigvee_{i=1}^{n} t_c^i$, where every t_c^i contains only qualified attributes w.r.t. $\mathcal{I}_Q$ or aggregated attributes. This step is performed only once.

We continue by performing the following procedures regarding one t_c of $unR(\mathcal{P})$ at a time, as indicated by the first two lines of Alg. 1. The union of the results produced for each t_c^i corresponds to the final Why-Not answer w.r.t. $\mathcal{P}$.

2a) CompatibleFinder. From t_c, we can easily compute the direct compatible set of tuples $Dir^{t_c} \subseteq \mathcal{I}_Q$ w.r.t. t_c, by performing appropriate SELECT statements that retrieve ids[2] of the relations referenced by the qualified attributes of t_c (as illustrated in Example 3.1). Note that, we demand that all (attribute:value) pairs in t_c that reference the same relation must co-occur in the same source tuple, also illustrated below. In parallel, $InDir^{t_c}$ is determined, as defined in the context of Def. 2.8.

2b) Canonicalize Q. A relational query Q may result in various equivalent query plans (trees) and similarly to [2, 5], we choose a canonical query tree representation that limits the equivalent query trees to consider. The following two rationales guide our choice of canonical query tree representation that differs from the canonical tree representation of [2].

First, we favor finding selections as Why-Not answers over finding joins, as selections are easier to inspect and change by a developer. Furthermore, this choice allows us to potentially reduce the runtime of *NedExplain*, since it allows us to push down selections (and as we shall see, we traverse and evaluate operators of the query tree in a bottom-up order).

Second, as described by Def. 2.12, we need to determine if a subquery (tree node) is picky and whether the condition $t_c.cond_\alpha$ is satisfied by the subquery's input and not in its output. In order to maximize the number of subqueries for which we can verify these conditions, we organize joins such that we obtain a view V of minimal query size where $type(V) \supseteq G \cup \{A_1, \ldots, A_n\}$ and no cross product is necessary. Intuitively, V corresponds to the subquery closest to the leaf level in the query tree joining all grouped and aggregated attributes. We refer to V as *breakpoint* subquery. Obviously, for queries without aggregation, the condition $type(V) \supseteq G \cup \{A_1, \ldots, A_n\}$ is trivially satisfied for any leaf node, i.e., for any $V \in Inst_Q$ (as $G \cup \{A_1, \ldots, A_n\} = \emptyset$), which results in all leaf nodes being breakpoint queries. Similarly, all leaf nodes representing relations in $\mathcal{I}_Q \setminus \mathcal{I}_V$ can be considered as breakpoint queries. We refer to the set of all breakpoint queries, i.e., $V \cup (\mathcal{I}_Q \setminus \mathcal{I}_V)$ as *visibility-frontier*. Given the query tree with

[2]These queries assume that each table has a key attribute to uniquely identify a tuple. The queries can however be trivially modified to SELECT * queries if no such key exists, processing will however take more space.

minimized breakpoint queries, we place the selections above and closest to the visibility-frontier to satisfy our first rationale.

Ex. 3.1 clarifies how we obtained the canonical query tree illustrated in Fig. 1 (c). Based on the appropriate selection of the visibility frontier at Q_2, the selection on $A.dob$ has been placed as close to this frontier as possible.

In the sequel, we denote our canonical query tree satisfying the above rationales as T.

2c) Primary global structure $\mathbf{Tab_Q}$. *NedExplain* relies heavily on one main global structure, denoted Tab_Q. Tab_Q stores intermediate computations as well, as discussed later. More specifically, Tab_Q contains the following *labeled entries* for each subquery m of Q.

- $Input$: the input tuple set of subquery m
- $Output$: the output tuple set of subquery m
- $Compatibles$: the set of tuples defined by

$$\{t_i \mid t_i \in m.Input \wedge (t_i \in Dir^{t_c} \vee t_i \text{ successor of some } t \in Dir^{t_c} \text{ w.r.t. } m)\}$$

- $Level$: the depth of m in T (the root having level 0)
- $Parent$: the parent node (subquery) of m in T
- Op: the root operation of m

To refer to the entry labeled l of a subquery m, we write $m.l$, e.g., $m.level$ refers to the level of subquery m.

Initialization is trivial for $m.Op$, $m.Parent$ and $m.Level$ based on T. For $m.Input$, initialization is possible for any m that is a base relation, based on $m.Input=\{\mathcal{I}_Q|_{R_i}\}$ *where* $m=[R_i], R_i \in \mathcal{S}_Q$. Then, we initialize $m.Compatibles$ for any m that is a base relation by $m.Compatibles=\{Dir^{t_c}|_{R_i}\}$ *where* $m = [R_i], R_i \in \mathcal{S}_Q$. The rest of the entries get updated during the execution of the algorithm. In order to efficiently access the information in Tab_Q that is necessary during processing, subqueries are stored in order of decreasing depth ($m.Level$) in the query tree. We access subquery m at position i using the notation $m = Tab_Q[i]$.

2d) Secondary global structures. Apart from Tab_Q, we make use of some other global structures, which are:

- $EmptyOutputMan$: the set of subqueries producing the empty set, used to determine the secondary Why-Not answer.
- $Non\text{-}PickyMan$: the set subqueries, producing successors of compatible tuples.
- $PickyMan$: the set of pairs (m, blocked), where m is a subquery and blocked=$\{t|t \in m.Input \wedge$ m is picky w.r.t. t and $Dir^{t_c} \cup InDir^{t_c}$ for $t\}$.

  This structure allows us to determine both the detailed and the condensed Why-Not answer.

EXAMPLE 3.1. *Given our running example and the c-tuple* $t_c=((A.name:Homer, ap:x_1), x_1>25)$, $CompatibleFinder$ *executes the SQL query* `SELECT A.aid FROM A WHERE A.name = 'Homer'` *to obtain* $Dir^{t_c} = \{t_4\}$. *Canonicalization of the query in Fig. 1(a) results in the query tree of Fig. 1(c), where the minimum subquery containing both* `A.name` *and* `B.price` *is the subquery* Q_2, *i.e.,* $V = (A \bowtie_{\{A.aid,AB.aid,aid\}} AB) \bowtie_{\{AB.bid,B.bid,bid\}} B$. *Here,* $\mathcal{I}_V = \mathcal{I}_A \cup \mathcal{I}_{AB} \cup \mathcal{I}_B$ *and* $\mathcal{I}_Q \setminus \mathcal{I}_V = \emptyset$, *so our visibility-frontier consists of* V *only. The selection operator* $\sigma_{A.dob>800BC}$ *is then placed just above* V*(i.e.,* Q_2*). Tab. 1 shows the initialization of* Tab_Q *given the canonical query tree of Fig. 1(c).*

Entry label	A	AB	$\mathbf{m_{Q_1}}$	B	$\mathbf{m_{Q_2}}$	$\mathbf{m_{Q_3}}$	$\mathbf{m_Q}$
Input	$\mathcal{I}_{\mid A}$	$\mathcal{I}_{\mid AB}$		$\mathcal{I}_B$			
Compatibles	$\{t_4\}$	$\emptyset$		$\emptyset$			
Output							
Level	4	4	3	3	2	1	0
Parent	m_{Q_1}	m_{Q_1}	m_{Q_2}	m_{Q_2}	m_{Q_3}	m_{Q_4}	
Op	relation schema	relation schema	$\bowtie$	relation schema	$\bowtie$	σ	α

Table 1: Primary global structure Tab_Q

Algorithm 1: *NedExplain*

Input: $\mathcal{S}_Q, Q, \mathcal{I}_Q, t_c$
Output: $Detailed$, the detailed Why-Not answer

```
1  CompatibleFinder;
2  Canonicalize(Q);
3  Initialization of global structures: Tab_Q, Non-PickyMan, EmptyOutputMan,
   PickyMan;
4  for (int i=0,..., Tab_Q.size()-1) do
5      m=Tab_Q[i];
6      if (checkEarlyTermination(m)) then
7          return DetailedAnswer();
8      m.Output← m(m.Input);
9      p←m.Parent;
10     p.Input←p.Input ∪ m.Output ;
11     if (m.Ouput=∅) then
12         EmptyOutputMan← EmptyOutputMan ∪{m};
13         if (m.Compatibles≠ ∅) then
14             PickyMan ← PickyMan ∪{m, m.Compatibles} ;
15     if (m.Op ≠ 'relation schema') then
16         p.Compatibles ← p.Compatibles ∪ FindSuccessors(m);
17     else
18         if (m.Compatibles≠ ∅) then
19             p.Compatibles ← p.Compatibles ∪m.Compatibles;
20             Non-PickyMan←Non-PickyMan∪{m};
21 return null;
```

3.2 Computing Why-Not Answers

After initialization, we visit the subqueries of Q in the order given by Tab_Q, which corresponds to the decreasing depth of subqueries in the query tree. At each step, we identify the successors of the compatible tuples and keep track of the picky and non-picky subqueries along the way. In the end, we return the Why-Not answer (due to the lack of space, we focus only on the detailed Why-Not answer here).

Alg. 1 is the main algorithm of *NedExplain*. It takes as input one unrenamed c-tuple t_c, so it will be executed once for each t_c in $UnR(\mathcal{P})$. The overall answer w.r.t. $\mathcal{P}$, will then be the union of all answers for each t_c.

Alg. 1 initializes the compatible tuple sets as well as the global structures w.r.t. t_c in lines 1–3. Next, it iterates through all the subqueries stored in Tab_Q until $checkEarlyTermination$ (Alg. 2) called at line 6 returns true or the last entry in Tab_Q has been reached. If $checkEarlyTermination$ returns true, we compute and return the detailed Why-Not answer. Otherwise, we continue with the evaluation of the current subquery m on its input (line 8) and maintain the entries of m's parent p in Tab_Q. We also maintain the secondary global structures EmpytOutputMan and PickyMan (lines 11–14). For all subqueries except for those that correspond to relation schemas (as identified by $m.Op$), $FindSuccessors$ (Alg. 3) finds possible successors of compatible tuples in the output of the current subquery and maintains the secondary global structures PickyMan and Non-PickyMan (line 16). Otherwise, p.Compatibles and the global structures update as described in lines 17–20. We now further discuss the sub-algorithms that Alg. 1 calls.

Check for early termination (Alg. 2). This step decides whether

Algorithm 2: *checkEarlyTermination*

```
   Input: m, a subquery
   Output: a boolean value
1  i←poisition of m in Tab_Q;
2  if (i≠0 ∧ m.Level≠Tab_Q[i-1].Level) then
3      int j=i-1;
4      while (j≥0∧Tab_Q[j].Level=Tab_Q[i-1].Level)) do
5          if (Tab_Q[j] ∈ Non-PickyMan) then
6              return FALSE;
7          j←j-1;
8      while (i < Tab_Q.size()) do
9          if (Tab_Q[i].Op='relation schema') then
10             return FALSE;
11         i ← i + 1;
12 else
13     return FALSE;
14 return TRUE;
```

or not we have all information in hand to compute our Why-Not answer. To realise this, *checkEarlyTermination* evaluates the position of the node representing the input subquery m in the query tree, in relation with the position of compatible tuples and their successors at other subqueries. More specifically, if m is the leftmost node at some level in the query tree T, we perform two checks: 1) we check if in the former level we have any NonPicky subqueries (lines 4–7) and if not, 2) we also check if among the rest of the subqueries there exists one with type 'relation schema' (lines 8–11). Intuitively, we stop Alg. 1 if, according to *checkEarlyTermination*, there are no more compatible tuples to trace from m on.

Algorithm 3: *FindSuccessors*

```
   Input: m, a subquery
   Output: the successors of tuples in m.Compatibles, in the output of m
1  compOrigins← {m'|(m'.Output ∩ m.Compatibles ≠
   ∅) ∧ (m'.parent = m)};
2  successors← ∅;
3  foreach o ∈ m.Output do
4      if lineage(o) ⊆ Dir^tc ∪ InDir^tc then
5          successors ← successors ∪(lineage(o) ∩ Dir^tc);
6  Blocked ← m.Compatibles \ successors;
7  if (successors ≠ ∅) then
8      Non-PickyMan ← Non-PickyMan ∪{m};
9  if (Blocked ≠ ∅ ∧ v not subquery of m) ∨
10 (V subquery of m ∧ α_{G,F}(m.Input) ⊨ t_c.cond_α
11 ∧α_{G,F}(m.Output) ⊭ t_c.cond_α ) then
12     PickyMan ←PickyMan ∪ {(m,Blocked)};
13 return successors;
```

Finding and managing successors (Alg. 3). For every subquery m we know the compatible tuples in its input, i.e., $m.Compatibles$, and the set of output tuples $m.Output$. Using this information, Alg. 3 finds the valid successors in $m.Output$ of tuples in $m.Compatibles$ by checking for each o∈$m.Output$ if its lineage is in $Dir^{tc} \cup InDir^{tc}$ (line 4).

Any valid successor is returned to Alg. 1 to be added in the parent's compatible input and in this case, we further add m to the set of Non-PickyMan subqueries (line 8). Lines 9 through 12 maintain the global structure PickyMan, adding the pair of subquery m and the set of compatible tuples in the input of m without a successor in the output of m, namely $Blocked$ (possibly empty), to PickyMan. The condition for adding this pair to PickyMan implements the conditions for our detailed missing answer as defined in Def. 2.12. This detailed answer can thus be trivially derived from PickyMan (details omitted due to space constraints).

m	m.Input	m.Output	m.Compatibles	m.Blocked
A	$\mathcal{I}_{\|A}$	$\mathcal{I}_{\|A}$	t_4	$\emptyset$
AB	$\mathcal{I}_{\|AB}$	$\mathcal{I}_{\|AB}$	$\emptyset$	$\emptyset$
m_{Q_1}	$\mathcal{I}_{\|A}, \mathcal{I}_{\|AB}$	$t_4 \bowtie t_7, t_4 \bowtie t_8,$ $t_5 \bowtie t_9$	t_4	$\emptyset$
B	$\mathcal{I}_{\|B}$	$\mathcal{I}_B$	$\emptyset$	$\emptyset$
m_{Q_2}	$t_4 \bowtie t_7, t_4 \bowtie t_8$ $t_5 \bowtie t_9, \mathcal{I}_B$	$t_4 \bowtie t_7 \bowtie t_2,$ $t_4 \bowtie t_8 \bowtie t_1,$ $t_5 \bowtie t_9 \bowtie t_3$	$t_4 \bowtie t_7,$ $t_4 \bowtie t_8$	$\emptyset$
m_{Q_3}	$t_4 \bowtie t_7 \bowtie t_2,$ $t_4 \bowtie t_8 \bowtie t_1,$ $t_5 \bowtie t_9 \bowtie t_3$	$\emptyset$	$t_4 \bowtie t_7 \bowtie t_2,$ $t_4 \bowtie t_8 \bowtie t_1$	$t_4 \bowtie t_7 \bowtie t_2,$ $t_4 \bowtie t_8 \bowtie t_1$
m_Q	$\emptyset$		$\emptyset$	

Table 2: Tab_Q after running *NedExplain* on our example

EXAMPLE 3.2. *Fig. 2 summarizes the results generated while executing* NedExplain *and is actually an abstraction of Tab_Q. For each new iteration of Alg. 1, i.e., for each subquery in Tab_Q, a new row is added to this table until the algorithm exits. For a clarification on the generated results, consider the following indicative cases:*

- *row 1(m=A): This row concerns the database instance A. Alg. 2 does not signal an early termination, since m is the first node in Tab_Q. Continuing in Alg. 1, we set $m.Output$=$m.Input$. The parent subquery is m_{Q_1}; so, $m_{Q_1}.Input$ and $m_{Q_1}.Compatibles$ get initialized with $m.Output$ and $m.Compatibles$, respectively. Moreover, since m contains compatible tuples it is classified as Non-PickyMan.*
- *row 6(m=m_{Q_3}): Alg. 1 filled the previous rows of the table in previous iterations, as well as the current row's $m.Input$ and $m.Compatibles$. The latter has been filled with the successors of t_4 (we show their how-provenance [8] to show their lineage, and leave it to the reader to verify that they are indeed valid successors). Alg. 2 does not signal an early termination, since m_{Q_2} (the only former level subquery) is not picky. So, Alg. 1 continues with the evaluation of m on $m.Input$ and fills the entries $m.Output$, and the parent's $m_Q.Input$ accordingly. Continuing with the call to Alg. 3, we conclude that m is a picky subquery and that it has blocked all the tuples in m.Compatibles ($m.Blocked$=$m.Compatibles$), which means that no successors have survived this subquery. At this stage, we also have Non-PickyMan=$\{A, AB, m_{Q_1}, B, m_{Q_2}\}$ and PickyMan=$\{(m_{Q_3}, \{t_4\})\}$.*
- *row 7(m=m_Q): In this row, $m.Input$ and $m.Compatibles$ got their values from the previous step. The call to Alg. 2 marks the early termination of the algorithm; m_Q is the first subquery having $m.Level$=0 and m_{Q_3}, which is the only subquery in the previous level, is a picky subquery. Moreover, there are no upper subqueries that could contain some compatible tuples. So, Alg. 1 terminates by computing the detailed Why-Not answer $\{t_4, m_{Q_3}\}$.*

Algorithm discussion. The worst case time complexity of *NedExplain* is in $O(|Q|(L + Out))$, $|Q|$ denotes the number subqueries of query Q, Out is the worst case size (in terms of number of tuples) of a subquery's output, and L is the height of the query tree. The number of detailed answers returned is bound by $|Dir^{tc}| + |Q| - |V|$, where $|V|$ is the number of subqueries in the breakpoint query V. We can also prove that *NedExplain* is correct and complete w.r.t. to the framework provided by our definitions, in the sense that it will return a pair $(t_{\mathcal{I}}, Q')$ for every compatible tuple $t_{\mathcal{I}}$ and a maximal set of pairs $(\perp, Q')$ due to our canonical tree representation. However, the subqueries returned as Q''s may vary for varying equivalent canonical query tree representations.

In the future, we will investigate algorithms to be invariant w.r.t. equivalent query tree rewritings.

4. EXPERIMENTS

In this section we display a comparative evaluation of our algorithm with respect to the Why-Not algorithm [2]. Briefly, the Why-Not algorithm [2] identifies a set of *frontier picky manipulations* that are responsible for the exclusion of missing-answers from the result by tracing *unpicked data items* (tuples) through the workflow. Two alternatives are proposed for traversing the workflow: a bottom-up approach and a top-down approach. The main difference between the two approaches lies in the efficiency of the algorithms (depending on the query and the Why-Not question). In [2], it is stated that both approaches are equivalent as they produce the same set of answers. We have implemented *NedExplain* and *Why-Not* (actually, its bottom-up version as it most resembles the approach of *NedExplain*) using Java, based on source code kindly provided by the authors of Why-Not. The original Why-Not implementation, as well as ours, relies on the lineage tracing provided by Trio (http://infolab.stanford.edu/trio/). We ran the experiments on an Oracle Virtual Machine running Windows 7 and using 2GB of main memory of a Mac Book Air with 1.8 GHz Intel Core i5, running MAC OS X 10.8.3. We used PostegreSQL 9.2 as database.

4.1 Use Cases

Our datasets originate from three databases named *crime*, *imdb*, and *gov*. The *crime* database corresponds to the sample crime database of Trio and was previously used to evaluate Why-Not. The data describes crimes and involved persons (suspects and witnesses). The *imdb* database is built on real-world movie data extracted from IMDB (http://www.imdb.com) and MovieLens (http://www.movielens.org). Finally, the *gov* database contains information about US congressmen and financial activities (collected at http://bioguide.congress.gov, http://usaspending.gov, and http://earmarks.omb.gov). The size of the relations in the databases ranges from 89 to 9341 records, with *crime* being the smallest and *gov* the largest database. For abbreviation, in the following discussion each relation instance is referred to by its initials, for example *M* refers to the Movies instance and *L* to the Locations instance. Moreover, when multiple instances of some relation are needed, we distinguish them by numbers, e.g., *M1* and *M2*.

For each database, we have created a series of use cases (see Tab. 4). Each use case consists of a query further defined in Tab. 3[3] and a Why-Not question in form of a predicate $\mathcal{P}$ as defined by Def. 2.6. The queries have been designed to include simple (Q4, Q6) and more complicated (Q1, Q3, Q5, Q7) queries, queries containing self-joins (Q3, Q4), queries having empty intermediate results (Q2), SPJA queries (Q8, Q9) and SPJU queries (Q12). To pinpoint the differences between the two algorithms, some use cases consider the same query with a different predicate.

Based on these use cases, we evaluate *NedExplain* and Why-Not, both in terms of answer quality and efficiency.

4.2 Answer Quality

Tab. 5 summarizes the why-not answers obtained by running our scenarios on Why-Not and *NedExplain* . For *NedExplain*, we distinguish among the detailed, the condensed and the secondary Why-Not answer, as defined by Defs. 2.12–2.14.

At first sight, the answers provided by Why-Not are simpler and clearer; they generally consist of a small number of subqueries.

[3]For easy of presentation, in presence of renaming, we display only the new attributes introduced by renaming.

Query	Expression
Q1	$\pi_{P.name,C.type}(C \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P)$
Q2	$\pi_{P.name,C.type}((\sigma_{C.sector>99}(C)) \bowtie_{sector1} W \bowtie_{witnessName} (S) \bowtie_{hair,clothes} P)$
Q3	$\pi_{W.name,C2.type}(W \bowtie_{sector2} C2 \bowtie_{sector1} \sigma_{C.type=Aiding}(C))$
Q4	$\pi_{P2.name}(\sigma_{P1.name \neq P2.name}(P2 \bowtie_{hair} (\sigma_{P1.name<B}(P1)))$
Q5	$\pi_{name,L.locationid}(L \bowtie_{movieId} ((\sigma_{M.year>2009}(M)) \bowtie_{name} (\sigma_{R.rating \geq 8}(R))))$
Q6	$\pi_{Co.firstname,Co.lastname}($ $(\sigma_{AA.party=Republican}(AA)) \bowtie_{id} (\sigma_{Co.Byear>1970}(Co)))$
Q7	$\pi_{SPO.sponsorId,SPO.sponsorln,E.camount}(E \bowtie_{earmarkId}$ $(\sigma_{ES.substage=Senate\ Committee}(ES) \bowtie_{id}$ $(\sigma_{SPO.party=Republican}(SPO)))$
Q8	$\alpha_{\{P.name\},\{count(C.type) \rightarrow ct\}}(\sigma_{sector>80}$ $(C \bowtie_{sector} W \bowtie_{witnessName} S \bowtie_{hair,clothes} P))$
Q9	$\alpha_{\{SPO.sponsorln\},\{sum(E.camount) \rightarrow am\}}($ $\sigma_{substage=Senate\ Committee}($ $\sigma_{party=Republican}(E \bowtie_{earmarkid} ES \bowtie_{id} SPO)))$
Q10	$\pi_{Co.lastname}(Co \bowtie_{id} ((\sigma_{AA.state=NY}$ $(\sigma_{AA.party=Democrat}(AA)))))$
Q11	$\pi_{SPO.sponsorln}(\sigma_{SPO.state=NY}$ $(\sigma_{SPO.party=Democrat}(SPO)))$
Q12	$Q10 \cup Q11$

Table 3: Use cases relational queries

Use case	Query	Predicate
Crime1	Q1	(P.Name:Hank,C.Type:Car theft)
Crime2	Q1	(P.Name:Roger,C.Type:Car theft)
Crime3	Q2	(P.Name:Roger,C.Type:Car theft)
Crime4	Q2	(P.Name:Hank,C.Type:Car theft)
Crime5	Q2	(P.Name:Hank)
Crime6	Q3	(C2.Type:kidnapping)
Crime7	Q3	(W.Name:Susan,C2.Type:kidnapping)
Crime8	Q4	(P2.Name:Audrey)
Crime9	Q8	((P.Name:Betsy,ct:x),x>8)
Crime10	Q8	(P.Name:Roger)
Imdb1	Q5	(name:Avatar)
Imdb2	Q5	(name:Christmas Story,L.locationId:USANew York)
Gov1	Q6	(Co.firstname:Christopher)
Gov2	Q6	(Co.firstname:Christopher,Co.lastname:MURPHY)
Gov3	Q6	(Co.firstname:Christopher,Co.lastname:GIBSON)
Gov4	Q7	(sponsorId:467)
Gov5	Q7	((SPO.sponsorln:Lugar,E.camount:x),x>=1000)
GOV6	Q9	((name:Bennett,am:x),x=18700)
Gov7	Q12	(name:JOHN)

Table 4: Use cases

On the other hand, *NedExplain* provides an answer more complex in structure, but more informative. Notice that the condensed answer resembles the answer returned by Why-Not and provides an "easily-consumable" answer as well. Still, they are not the same, as we will describe later in this section. The subsequent discussion first highlights the differences of the various kinds of *NedExplain* answers before we compare the results of *NedExplain* with those that Why-Not produces. Due to the lack of space, we limit the discussion to representative use cases.

Detailed vs. condensed and secondary Why-Not answers. Consider the *Crime6* use case associated to the query tree for $Q3$ depicted in Fig. 4(b). The condensed answer (but not the Why-Not algorithm answer, as described later) indicates that m_8 is a picky subquery. The detailed answer consists of pairs of the form $(cd_i, m_8), i = 1, \ldots, 11$. For this case, the detailed answer does not provide substantial new insights compared to the condensed answer, as m_8 is responsible for pruning all compatible tuples. In this situation, the condensed answer is the most appropriate to return to the user.

However, in other cases, the simplicity of the condensed answer

Use case	Why-Not	*NedExplain* Answers: Detailed	Condensed	Sec.
Crime1		$(P.Id{:}2, m_2)$,$(C.Id{:}2, m_2)$	m_2	
Crime2	m_0	$(P.Id{:}604, m_0)$,$(C.Id{:}2, m_2)$	m_0, m_2	
Crime3	m_0,m_4	$(P.Id{:}604, m_0)$,$(C.Id{:}2, m_4)$	m_4, m_0	
Crime4	m_4	$(P.Id{:}2, m_5)$,$(C.Id{:}2, m_4)$	m_4, m_5	
Crime5	m_4	$(P.Id{:}2, m_5)$	m_5	m_4
Crime6	m_7	$(C2.Id{:}396, m_8)$, $(C2.Id{:}85, m_8)$, $\ldots$, $(C2.Id{:}112, m_8)$	m_8	
Crime7	m_7	$(C2.Id{:}396, m_8)$, $(C2.Id{:}85, m_8)$, $\ldots$, $(C2.Id{:}112, m_8)$, $(W.Id{:}2, m_9)$	m_8, m_9	
Crime8		$(P2.Id{:}51, m_{12})$	m_{12}	
Crime9	n.a.	$(null, m_3)$	m_3	
Crime10	n.a.	$(P.Id{:}604, m_0)$	m_0	
Imdb1	m_1	$(R.Id{:}124, m_2)$, $(M.Id{:}18, m_1)$	m_1,m_2	
Imdb2		$(L.Id{:}1, m_3)$, $(M.Id{:}4, m_3)$, $(R.Id{:}245, m_3)$	m_3	
Gov1	m_2	$(Co.Id{:}569, m_0)$, $(Co.Id{:}1495, m_0)$, $(Co.Id{:}1072, m_2)$, $(Co.Id{:}772, m_0)$	m_0, m_2	
Gov2	m_1	$(Co.Id{:}1072, m_2)$	m_2	
Gov3	m_0	$(Co.Id{:}569, m_0)$	m_0	
Gov4	m_4	$(SPO.Id{:}9, m_4)$, $(ES.Id{:}80, m_8)$, $(ES.Id{:}78, m_8)$, $(ES.Id{:}79, m_8)$	m_4, m_8	
Gov5	m_6	$(E.Id{:}15, m_6)$, $(E.Id{:}324, m_6), \ldots,$ $(E.Id{:}533, m_6)$, $(SPO.Id{:}199, m_6)$	m_6	
Gov6	n.a.	$(null, m_7)$	m_7	
Gov6	n.a.	$\{(Co.Id{:}772, m_{11})\}, \{\}$	$\{m_{11}\}$,$\{\}$	

Table 5: Why-Not and *NedExplain* answers, per use case

may hide from the user more specific, but essential information, e.g., in cases where the answer is not a single subquery. For instance, in *Crime7*, the condensed answer (again not the same as in Why-Not as explained later) identifies m_8 and m_9 as picky subqueries (refer to $Q3$ in Fig. 4(b)). From the detailed answer, we moreover obtain the knowledge that there were eleven tuples (originating from Co) for which m_8 is picky, but also one tuple (originating from W) for which m_9 is picky. This information can be useful, as it indicates that no valid successors of compatible crime tuples reached m_9 to join with valid witness tuples. So, the existence of a more detailed answer can be of major help towards understanding the "background" of the result.

***NedExplain* vs. Why-Not.** Our first comparison between *NedExplain* and Why-Not focuses on the $Crime5$ use case, with the associated query $Q2$ (its query tree is given in Fig. 4(a)) having an intermediate empty result on m_4 ($\sigma_{sector>99}(C)$). The Why-Not algorithm identifies m_4 as a picky subquery, which is correct in the sense of responsibility for the missing result tuple. Yet, it cannot be considered as picky in the strict form, since it is not blocking directly any compatible tuple. In this use case, *NedExplain* provides a more complete and descriptive answer. It identifies m_5 as a picky subquery, and in addition it includes m_4 in the secondary answer. Knowing that m_4 produced an empty set right before the join in m_5 can possibly be another reason for m_5 being identified as picky.

Let us now focus on the cases where the answers of Why-Not and *NedExplain* differ, i.e., $Crime6$ and $Crime7$. Both use cases relate to $Q3$, which contains a self join on relation Crime. The Why-Not algorithm falsely identifies m_7 ($\sigma_{C1.type=Aiding}(C1)$) as a picky subquery, because it locates the compatible tuples in both $C1$ and $C2$. As a result the compatible tuples from $C1$ with $type{:}Kidnapping$ are naturally picked at m_7. This problem is solved by our algorithm, by introducing the notion of qualified attributes. In this way, we locate the compatible tuples only in the correct instance of the relation Crime, i.e., $C2$, according to the type of the output of the query $Q3$.

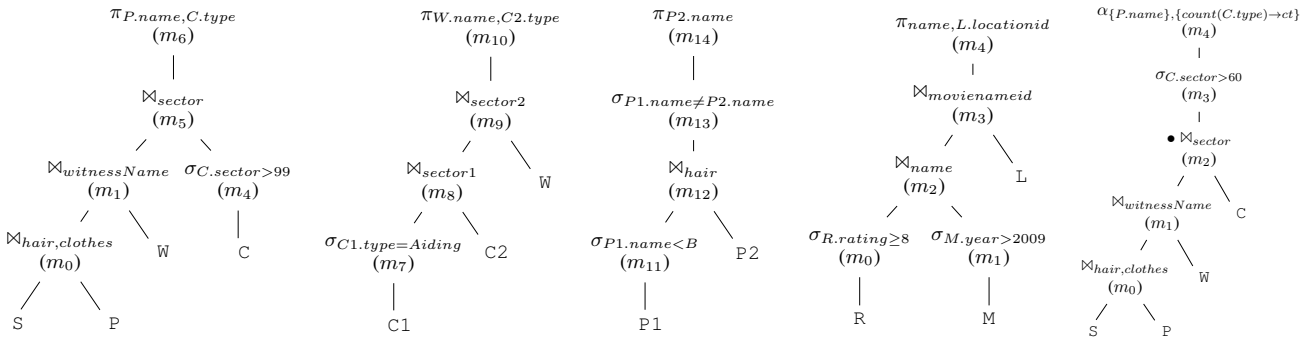
Another problem having its origin in the identification of compatible tuples can be spotted at use case $Crime8$. Even though it is based on a very simple query (refer to $Q4$ in Fig. 4(c)), the Why-Not algorithm finds no answers at all. $Q4$ searches for persons that have the same hair as persons whose names start with a letter smaller than B (while not being the same person). The Why-Not algorithm places its compatible source tuples both in P1 and P2. The one coming from P2 does not find any join partners in m_{12} from P1 as the only candidate ones have names starting with C or D, namely Davemonet, Chiardola, and Debye. So m_{12} is picky for the compatible tuple coming from P2. The one coming from P1 survives the selection of m_{11} and joins with the three persons with equal hair color coming from P2, namely Davemonet, Chiardola, and Debye. Hence, m_{12} is not picky for three successors of the compatible tuple originating from P1, and it is easy to verify that the same is true for the remaining subqueries to be processed. Hence, Why-Not believes that Audrey is actually not missing from the result.

NedExplain on the other hand will correctly locate the compatible tuple only in P2. As mentioned previously, all candidate join partners coming from P1 for Audrey (that comes from P2), will be discarded in m_{11}, making m_{12} a picky subquery for the associated compatible tuple ($P2.Id{:}51$).

Next, we review the $Imdb2$ use case where the associated predicate $\mathcal{P}$ is not in its unrenamed form. This means that it contains attributes that are not in $\mathcal{S}_Q$, but instead were introduced through renamings associated to the subqueries. This use case is based on query $Q5$ (see Fig. 4(d)), with the following renaming associated to its subquery m_2 (join): $\nu = (R.moviename, M.moviename, name)$. The considered predicate $\mathcal{P}$=(name:Christmas Story,L.locationId:USANew York) refers to the new attribute $name$ associated with the subquery $m_0 \bowtie m_1$ in $Q5$. Thus, before running *NedExplain*, we transform $\mathcal{P}$ to its unrenamed form: (R.name: Christmas Story, L.locationId: USANew York) $\bowtie$ (M.name: Christmas Story, L.locationId: USANew York) = (R.name: Christmas Story, M.name: Christmas Story, L.locationId: USANew York). As we said, for the compatible tuples we calculate *valid successors* to trace in the query tree. This method leads us to the identification of m_3 as a picky subquery, i.e. the subquery after which we find no more valid successors. Why-Not on the contrary relies on tracing successors (not necessarily valid) of the compatible tuples, which in this case still can be found in the result set. So, no picky subqueries are identified and no answers are returned.

Let us now focus on $Crime9$, based on the SPJA query $Q8$ (see Fig. 4(e)). As explained in Sec. 3, *NedExplain* identifies as breakpoint - marked as a bullet on the tree representation - the subquery $m2$ and the operators of the query are executed in a predefined order (putting selections as near as possible to the breakpoint subquery). In this way, we are able to identify $m3$ as a why-not answer; the condition of the predicate of this scenario, $ct > 8$ is satisfied in the input pf $m3$ (am=13) but not in its output (ct=7). Thus, the detailed answer is $m3$ associated with a null value, since $m3$ is not picky in a strict sense (valid successors of $P.id{:}604$ still exist in the output of $m3$). A variation of this use case is the use case $Crime10$, where we obtain the detailed answer $P.id{:}604, m_0$, now bound to a specific tuple id, as m_0 completely erases the trace of $P.id{:}604$. Note that Why-Not results are not available (n.a.) as the implementation provided by the authors does not support aggregation and from [2], it is unclear how compatible tuples or the Why-Not question are defined in this case.

Overall, in this first study on answer quality, we observe that *NedExplain* produces a correct answer for all use cases, as opposed to Why-Not that showcases no, imprecise, or incomplete results in



(a) Query $Q2$ (Crime3, Crime4, Crime5)
(b) Query $Q3$ (Crime6, Crime7)
(c) Query $Q4$ (Crime8)
(d) Query $Q5$ (Imdb1, Imdb2)
(e) Query $Q8$ (Crime9, Crime10)

Figure 4: Query trees for queries $Q2$, $Q3$, $Q4$, $Q5$, $Q8$

some cases. Furthermore, the different types of answers *NedExplain* returns convey more information than answers returned by the Why-Not algorithm, potentially improving a developer's analysis and debugging experience in the context of relational queries. To summarize the highlighted shortcomings, we present the following list:

Inaccurate selection of unpicked data. The selection of the source data items to trace in the workflow does not take into account self-joins, possibly leading to no explanation or a wrong explanation. Furthermore, the authors choose not to trace data that contributed to some result (i.e., data in the lineage of any result tuple), which also reduces the set of why-not questions for which an explanation may be returned. *Crime7* and *Crime8* are use cases illustrating this.

Insufficient definition of successors. The source data items of interest are traced independently from each other, not considering the global picture given by the occurrence of other source data items in the trace line, and which we model as *valid* successors. In this way, inaccurate manipulations are found to be responsible (or not responsible) for the missing-answers, as use cases *Crime5* and *IMDB2* showcase.

Restriction to frontier picky manipulations. Partial results are computed by the decision to return only $frontier$ picky manipulations, which strongly depend on the query tree representation. For instance, refer to use cases *Crime2* and *IMDB1*.

Insufficient answer detail. Why-Not may return more than one answer (i.e., a set of manipulations), in which case it is difficult to tell the contribution of each of the returned manipulations. Use cases *Crime3* and *GOV4* demonstrate this lack of detail.

We defer a more thorough user study to future work.

4.3 Runtime Evaluation

Considering runtime, we first study the runtime distribution over the different phases of *NedExplain*. We then compare *NedExplain*'s runtime to the runtime of Why-Not.

Phase-wise runtime. We distinguish among four phases of *NedExplain*: (1) *Initialization*: the global structures initialization; (2) *CompatibleFinder*: the computation of the compatible tuples set; (3) *SuccessorsFinder*: the computation of successors of compatible tuples at each subquery output, corresponding to Alg. 3; and (4) *Bottom-up traversal*: the traversal of the query tree (including SQL query executions) following the bottom-up approach, i.e. Alg. 1 without initialization and call to Alg. 3.

Fig. 5 reports the distribution of the execution time over these phases for each of our use cases. First, we observe a similar distribution for use cases referring to the same query, e.g., *Crime1*

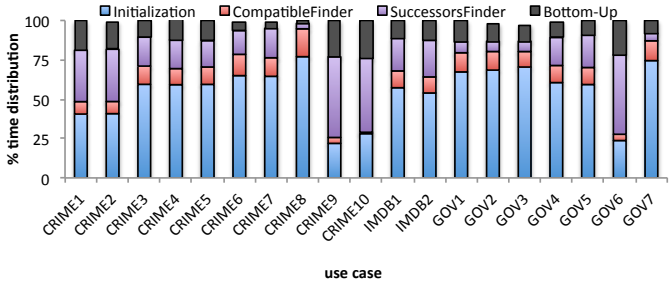


Figure 5: Phase-wise runtime for *NedExplain*

and *Crime2* or *GOV1*, *GOV2*, and *GOV3*, due to the fact that these use cases engage the same database instances and evaluate (almost) the same operators. For use cases involving different queries, we observe that in general, for SPJ queries, the overall runtime is dominated by the initialization phase (between 40 and 77%), essentially caused by the initialization of all relevant java objects. After initialization, SuccessorFinder has the second largest impact on overall runtime for most SPJ use cases (*Gov1 – Gov3*, all based on $Q6$, and *Crime8* being the exceptions). This phase essentially corresponds to Alg. 3, where we compute lineage and do set comparisons. Focusing on the *Crime8* exception use case, the last valid successor is lost very early (on m_{12} in Fig. 4(c)) after evaluating simple operators, which explains both the low fraction of runtime used to find successors but also the low time for the bottom-up traversal. The picture changes when considering SPJA queries, where most of the time is dedicated to the computation of valid successors. This can be explained, by the extra computations needed in the SPJA case of Alg. 3. These computations basically require additional SQL query executions, on the input and output of tree nodes placed after the breakpoint.

Runtime comparison to Why-Not. Fig. 6 displays, for each use case, the time (in ms) each algorithm needs to produce its Why-Not answers. Generally, we observe that *NedExplain* is faster compared to Why-Not. One reason is that the implementation of the Why-Not algorithm requires the usage of Trio for lineage calculation, adding a substantial overhead to runtime especially when many trio tables are referenced as in $Crime1$ and $Crime2$. *NedExplain* traces the compatible tuples by issuing queries directly to the underlying Postgres database based on their unique identifiers in order to find their successors, which speeds up the process.

In the future, we plan to more extensively study the impact of

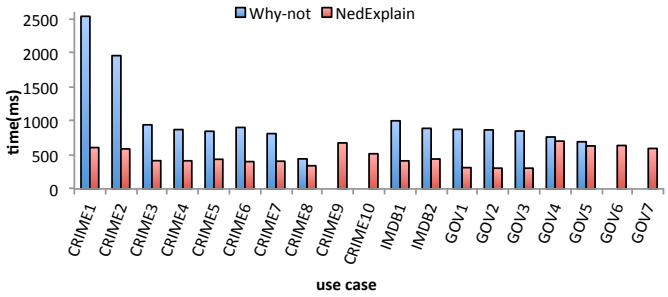


Figure 6: Why-Not and *NedExplain* execution time

various parameters on runtime. To conclude, this first set of experiments on runtime indicates that *NedExplain* can provide high-quality Why-Not answers in reasonable time.

5. CONCLUSIONS AND OUTLOOK

We have addressed the issue of answering *why-not* questions by first formally defining, for the first time, the domain and concepts of *Why-Not* questions and *Why-Not* answers w.r.t. relational queries including projection, selection, join, union, and aggregation. Based on these definitions, we have described *NedExplain*, an algorithm to produce correct Why-Not answers. As discussed and validated through experiments, *NedExplain* is capable of providing a more complete and correct set of answers, compared to the state-of-the-art algorithm, while being competitive in terms of runtime. This set of query based explanations could further be used to obtain modification-based explanations and/or in combination with instance based explanations to calculate *hybrid* explanations to Why-Not questions.

In the future, besides a more thorough experimental study of algorithm behavior, we plan to extend our algorithm to also consider set difference. This would require tracing data that needs to make it to the result (compatible tuples) but also data that should not make it (in order not to eliminate the compatible tuples). We further plan to study the issue of computing why-not answers such that the result is invariant w.r.t. equivalent logical query rewritings.

6. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] A. Chapman and H. V. Jagadish. Why not? In *International Conference on the Management of Data (SIGMOD)*, pages 523–534, 2009.

[3] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4), 2009.

[4] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *International Conference on Data Engineering (ICDE)*, pages 367–378, 2000.

[5] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179 – 227, 2000.

[6] J. Danaparamita and W. Gatterbauer. QueryViz: helping users understand SQL queries and their patterns. In *International Conference on Extending Database Technology (EDBT)*, pages 558–561, 2011.

[7] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[8] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Principles of Database Systems (PODS)*, pages 31–40, 2007.

[9] T. Grust and J. Rittinger. Observing sql queries in their natural habitat (preprint). *ACM Transactions on Database Systems*, 0(0), 2012.

[10] Z. He and E. Lo. Answering why-not questions on top-k queries. In *International Conference on Data Engineering (ICDE)*, pages 750–761, 2012.

[11] M. Herschel. Wondering why data are missing from query results? ask conseil why-not (short paper, to appear). In *International Conference on Information and Knowledge Management (CIKM)*, 2013.

[12] M. Herschel and H. Eichelberger. The Nautilus Analyzer: understanding and debugging data transformations. In *International Conference on Information and Knowledge Management (CIKM)*, pages 2731–2733, 2012.

[13] M. Herschel and T. Grust. Transformation lifecycle management with nautilus. In *VLDB Workshop on the Quality of Data (QDB)*, 2011.

[14] M. Herschel and M. A. Hernández. Explaining missing answers to SPJUA queries. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1):185–196, 2010.

[15] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):736–747, 2008.

[16] T. Imieliński and J. Witold Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.

[17] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. SnipSuggest: Context-aware autocompletion for SQL. *Proceedings of the VLDB Endowment (PVLDB)*, 4(1):22–33, 2010.

[18] A. Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *Proceedings of the VLDB (PVLDB)*, 4(12):1466–1469, 2011.

[19] J. Silva. Debugging techniques for declarative languages: Profiling, program slicing and algorithmic debugging. *AI Communications*, 21(1):91–92, 2008.

[20] Q. T. Tran and C.-Y. Chan. How to ConQueR why-not questions. In *International Conference on the Management of Data (SIGMOD)*, pages 15 – 26, 2010.